This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images,
please do not report the images to the
Image Problems Mailbox.

REMARKS/ARGUMENTS

Overview of the Office Action

Claims 1, 6, 7, 11, 14-15, and 21-22 have been rejected by the Examiner under 35 U.S.C. § 102(e) as being anticipated by Wugofski (U.S. Patent No. 6,317,143).

Claims 2-5, 8-10, 12-13, 16-20, and 23 have been rejected by the Examiner under 35 U.S.C. § 103(a) as unpatentable over Wugofski (U.S. Patent No. 6,317,143) in view of Buxton et al. (U.S. Patent No. 6,469,714).

Status of the Claims/Amendments

Claims 1-23 are pending.

Claims Rejected Under 35 U.S.C. § 102(e)

Claims 1, 6, 7, 11, 14-15, and 21-22 have been rejected by the Examiner under 35 U.S.C. § 102(e) as being anticipated by Wugofski (U.S. Patent No. 6,317,143). In response—and without conceding the validity or appropriateness of the Examiner's rejections—Applicants have submitted herewith, as Attachment A hereto, a Declaration Swearing Back of Reference Pursuant to 37 C.F.R. 1.131 with regard to the Wugofski reference. In light of this Declaration, Applicants respectfully submit that the rejection of Claims 1, 6, 7, 11, 14-15, and 21-22 has been traversed, and Applicants request that these claims be allowed to immediately issue.

Claims Rejected Under 35 U.S.C. § 103(a)

Claims 2-5, 8-10, 12-13, 16-20, and 23 have been rejected by the Examiner under 35 U.S.C. § 103(a) as unpatentable over Wugofski (U.S. Patent No. 6,317,143) in view of Buxton et al. (U.S. Patent No. 6,469,714). In response—and without conceding the validity or appropriateness of the Examiner's rejections—Applicants have submitted herewith, as Attachment A hereto, a Declaration Swearing Back of Reference Pursuant to 37 C.F.R. 1.131 with regard to the Wugofski reference. In light of this Declaration, and given that the Buxton reference alone is insufficient to anticipated the claimed invention of the present Application, Applicants respectfully submit that the rejection of Claims 2-5, 8-10, 12-13, 16-20, and 23 has been traversed, and Applicants request that these claims be allowed to immediately issue.

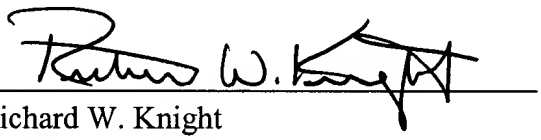
[Remainder of Page Intentionally Left Blank]

CONCLUSION

Based on the Declaration submitted herewith, Applicants respectfully submit that the Examiners rejections have been traversed and, accordingly, Applicants request that the rejections be withdrawn and that the claims be allowed to issue. Should the Examiner have any questions, comments, or suggestions that would expedite the prosecution of the present case to allowance, Applicants' undersigned representative earnestly requests a telephone conference at (206) 332-1394.

Respectfully submitted,

Date: June 8, 2004



Richard W. Knight
Registration No. 42,751

Woodcock Washburn LLP
One Liberty Place - 46th Floor
Philadelphia PA 19103
Telephone: (215) 568-3100
Facsimile: (215) 568-3439

ATTACHMENT A

Attachment immediately follows this page.

[Remainder of Page Intentionally Left Blank]

EXHIBIT A

Exhibit immediately follows this page.

[Remainder of Page Intentionally Left Blank]

8 2004 JC48

Adding Menus & Menu Items to the New Visual Studio Shell

vsmenu.doc[1]
2-Mar-99, 6:35 PM
Steve Seixeiro

RECEIVED
JUN 1 6 2004
Technology Center 2100

1. INTRODUCTION 2

2. MAKING MENUS 2

2.1 THE MENU ARCHITECTURE 2
2.2 COMMAND TABLE FORMAT 3
2.2.1 CMDS_SECTION – CMDS_END 3
2.2.1.1 MENUS_BEGIN – MENUS_END 4
2.2.1.2 NEWGROUPS_BEGIN – NEWGROUPS_END 5
2.2.1.3 BUTTONS_BEGIN – BUTTONS_END 5
2.2.1.4 COMBOS_BEGIN – COMBOS_END 7
2.2.1.5 BITMAPS_BEGIN – BITMAPS_END 8
2.2.2 CMDPLACEMENT_SECTION – CMDPLACEMENT_SECTIONEND 9
2.2.3 VISIBILITY_SECTION – VISIBILITY_SECTIONEND 9
2.2.4 KEYBINDINGS_SECTION – KEYBINDINGS_SECTIONEND 10
2.3 USING THE BINARY COMMAND TABLE 10

3. ROUTING COMMANDS TO YOUR PACKAGE 11

3.1 IOLECOMMANDTARGET INTERFACE 11
3.1.1 QueryStatus 11
3.1.2 Exec 12
3.1.3 Sample QueryStatus and Exec implementations 12
3.2 HOW COMMANDS ARE ROUTED TO YOUR PACKAGE 14

4. UPDATING THE SHELL'S UI 15

5. VB, VJ AND SHELL SPECIFIC FILES 15

6. RESOURCES 16

[1] This document's primary location is an SLM project on \\Hazelnut\VbDev in the *RadBin* project, *Doc* subdirectory.

1. Introduction

This document will outline the process you need to go through if you want to add items to the new shell, either as a package that plugs into the shell or items in the shell itself. It will also touch on the basic architecture of how menus work, but won't go into great detail about this (see the CommandBar UI spec on http:\\rad\specs for more in depth information). Finally, it will explain how these menu commands are routed to different packages.

2. Making Menus

Creating menus and menu items has come a long way from how it used to be done. This section will describe the new menu architecture that is used in the shell, as well as how to implement a command table that the shell will be able to read. This architecture allows different packages to contribute to the shell's new menu space in an easy way, without having to deal with the command bars directly.

2.1 *The Menu Architecture*

The new menu architecture is built around easily reusing and sharing commands between components and the shell itself. The shell employs a "Command Groups" approach, where commands are bunched together and end up belonging to particular groups. The easiest way to identify a group is as a set of commands or menus that exist between separators on a menu or toolbar (excluding the main menu bar). Groups are set up by the shell, but can also be created and placed by any other package. The shell has already defined a number of default groups that packages can add new commands or menus to.

One example of Command Groups that shows the concept's flexibility is the Cut/Copy/Paste group, defined by the shell. The shell has set up a group, call it IDG_VS_CUTCOPY, to which it has added the Cut, Copy, and Paste command. It then placed this group on the Edit menu. Now that the group has been defined, anybody can use it without having to redefine it, or any of the commands that are on it. This has the advantage of allowing easy access to the shell's commands to any other package. Say, for example, that a forms package comes along and wants to create a context menu that comes up on the form or its controls. Perhaps the context menu should include the Cut/Copy/Paste commands. Since this group has already been defined, the package doesn't need to add each of these items to its context menu, but would only have to add the group IDG_VS_CUTCOPY instead. This could be done over and over, in multiple packages – saving packages from redefining the items and their locations. Also, if a package or the shell added the Delete command to this group, then everywhere the group was used would now include the new command automatically.

In essence, to add commands to the new shell requires that a package follow the steps outlined below:

1. Define a group or find an existing group.
2. Define the commands that should be in this group or use existing commands.
3. Place the commands in the group.
4. Place the group on an existing menu or toolbar, or create a new menu or toolbar to add the group to.

To do this, the shell requires that the package registers a set of meta data that describes the package's menus, items and groups. This meta data should be stored as a binary resource in the package. The data is generated from a text file that is sent through a parser, which outputs the binary resource that the package should include. The next section will go through what should be in the text file. To find out more about the high level architecture regarding menus, please see the CommandBar UI document on http:\\rad\specs, under the Vegas UI section.

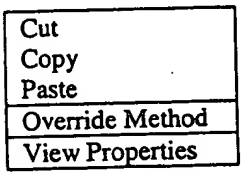
2.2 Command Table Format

The Command Table is basically a text file that contains sections that can be understood by a special parser. The file is first sent through the C preprocessor which allows including of other files (such as a file containing the definitions of commands, groups and menus). The parser is CTC.EXE (Command Table Compiler), and is currently located in the Vegas source tree, although it will eventually move to a more common place. CTC takes the text file as a parameter and generates a binary file from it. The resulting binary file, with the extension of .CTO, should be what the package includes as part of its resources.

This describes the basics of what the text file should look like, based on the sections within it. One common idea throughout the file is that menus, commands and groups are identified by both a GUID and a LONG so that they are unique. In general, each package that wants to add commands should create 2 GUIDs, one to be used for it's groups and menus and the other to be used by its commands.

2.2.1 CMDS_SECTION – CMDS_END

This section of the command table has five subsections, Menus, Groups, Buttons, Combos and Bitmaps. Each section describes the commands, groups or menus that should be created. There are different parameters for each section, which we'll get into next. We'll provide an example in each section which will in the end produce a context menu that looks like the following:



In addition, the CMDS_SECTION takes a GUID on its identifying line. This GUID should generally be the CLSID of the package that is registered with the shell. This enables the shell to know which commands were contributed by which packages. This is required when the package adds menu items to the shell, but isn't loaded when the shell starts up. If a command is selected that belongs to a package that is not yet loaded, the shell will then load that package.

The forthcoming pages will outline what parameters are necessary for each section, with the example in mind. An overview of the example as a whole would be as follows:

```
CMDS_SECTION GuidVJPackage

   MENUS_BEGIN
      GuidVJGrp:IDM_MYCONTEXT, GuidVJGrp:0, 0x0000, CONTEXT, "MyContextMenu", "&MyMenu";
   MENUS_END

   NEWGROUPS_BEGIN
      GuidVJGrp:IDG_CUTCOPY,        GuidVJGrp:IDM_MYCONTEXT,  0x0100;
      GuidVJGrp:IDG_OVERRIDE,       GuidVJGrp:IDM_MYCONTEXT,  0x0300;
      GuidVJGrp:IDG_PROPERTIES,     GuidVJGrp:IDM_MYCONTEXT,  0x0500;
   NEWGROUPS_END

   BUTTONS_BEGIN
      GuidVJCmd:VJCmdCut,        GuidVJGrp:IDG_CUTCOPY,     0x0100, GuidOfficeIcon:msotcidCut,        BUTTON, , "Cut";
      GuidVJCmd:VJCmdCopy,       GuidVJGrp:IDG_CUTCOPY,     0x0200, GuidOfficeIcon:msotcidCopy,       BUTTON, , "Copy";
      GuidVJCmd:VJCmdPaste,      GuidVJGrp:IDG_CUTCOPY,     0x0300, GuidOfficeIcon:msotcidPaste,      BUTTON, , "Paste";
      GuidVJCmd:VJCmdOverride,   GuidVJGrp:IDG_OVERRIDE,    0x0100, GuidOfficeIcon:msotcidNoIcon,     BUTTON, , "Override Method";
      GuidVJCmd:VJCmdProperties, GuidVJGrp:IDG_PROPERTIES,  0x0100, GuidOfficeIcon:msotcidProperties, BUTTON, , "View Properties", "&Properties";
   BUTTONS_END

CMDS_END
```

For a full featured sample file, you can look at the VSSHLPRJ.H file in the V6 tree. This is the file the Shell uses to determine the items and menu structure it provides for everyone else.

2.2.1.1 MENUS_BEGIN – MENUS_END

This is the section in which a package can define a new menu, toolbar or context menu. Each item has 8 parameters associated with it, all parameters being separated by a comma (end of line is denoted by a semicolon). They are, in order:

1. **GUID:MENUID**
   Ex. GuidVJGrp:IDM_MYCONTEXT
   This is the unique identifier of this menu.

2. **GUID:GroupId**
   Ex. GuidVJGrp:0
   This is the unique identifier of the Parent Group of this menu (ie where this group will primarily be placed). Context menus don't have parents so you can just use 0 as the GroupId.

3. **Priority**
   Ex. 0x0000
   The priority section is quite important. **You should only use the top TWO numbers when defining the priority for a command, group or menu.** This is required by the shell to deal with merging of items. So, a typical priority would be something like 0x0400 up to a maximum of 0xFF00. This parameter determines where the item is to be placed in the group specified in parameter 2, 0x0000 being the highest, and 0xFF00 being the lowest. Since context menus aren't really placed anywhere, simply put 0x0000 here.

4. **Type**
   Ex. CONTEXT
   This parameter determines the type of menu being created. The default, signified by leaving the field empty, is a normal menu. Other options are either CONTEXT or TOOLBAR.

5. **Button Text**
   Ex. "MyContextMenu"
   This parameter is the textual name of the menu if it were to appear as an item somewhere, either in the customize dialog or on the menu itself.

6. **Menu Text**
   Ex. "&MyMenu"
   This parameter is the textual name that would show up when the item is a menu on the main menubar.

7. **Tooltip Text**
   Ex. ,
   This parameter would be the tooltip that would show up if the item could display tooltips. Some menus can! However, if at any time after adding the Button Text, the text should be the same then you don't need to specify it any more. If fields 6, 7 and 8 were left blank, then the text for these items would be the same as the Button Text.

8. **Command Text**
   Ex. ;
   This parameter determines the text that is used by the command bar object model and extensibility to be able to get to this particular menu.

2.2.1.2 NEWGROUPS_BEGIN – NEWGROUPS_END

This is the section in which a package can define its groups. Each item has 4 parameters associated with it. They are, in order:

1. **GUID:GroupId**
   Ex. GuidVJGrp:IDG_CUTCOPY
   GuidVJGrp:IDG_OVERRIDE
   GuidVJGrp:IDG_PROPERTIES
   This defines the new group that you want to create.

2. **GUID:MenuId**
   Ex. GuidVJGrp:IDM_MYCONTEXT
   GuidVJGrp:IDM_MYCONTEXT
   GuidVJGrp:IDM_MYCONTEXT
   This defines the primary parent menu the group will be placed on.

3. **Priority**
   Ex. 0x0100
   0x0300
   0x0500
   This defines the priority of the group on the specified menu.

4. **Flags**
   Ex. None
   The only available flag is DYNAMIC. A group can be set to be DYNAMIC if it should be filled in by multiple components, or the number of items in it is unknown until run time. Most groups should NOT have this flag set.

2.2.1.3 BUTTONS_BEGIN – BUTTONS_END

This is the section in which a package defines its commands. A command, be it a toolbar button, menu item, or a split dropdown, is recognized as a simple button. For a list of possible commands, see the Command Bar UI spec on http:\\rad\specs under the types of commands section. The BUTTON type is the most common one used, since it applies to toolbars, menus and context menus. This section is comprised of 10 parameters:

1. **GUID:CmdId**
   Ex. GuidVJCmd:VJCmdCut
   GuidVJCmd:VJCmdCopy
   GuidVJCmd:VJCmdPaste
   GuidVJCmd:VJCmdOverride
   GuidVJCmd:VJCmdProperties
   This defines the new command you are creating.

2. **GUID:GroupId**
   Ex. GuidVJGrp:IDG_CUTCOPY
   GuidVJGrp:IDG_CUTCOPY
   GuidVJGrp:IDG_CUTCOPY
   GuidVJGrp:IDG_OVERRIDE
   GuidVJGrp:IDG_PROPERTIES
   This defines the Primary group that this command belongs to. Every command MUST belong to a primary group. This is necessary for both Keyboard and Toolbar Customization modes.

The command will show up in its primary location when the end user looks in either one of these dialogs.

3. **Priority**
   Ex. 0x0100
   0x0200
   0x0300
   0x0100
   0x0100
   This defines the priority of the command in the specified group.

4. **GUID:IconId**
   Ex. GuidOfficeIcon:msotcidCut
   GuidOfficeIcon:msotcidCopy
   GuidOfficeIcon:msotcidPaste
   GuidOfficeIcon:msotcidNoIcon
   GuidOfficeIcon:msotcidProperties
   This defines the icon that is associated with the command. By specifying GuidOfficeIcon, the shell will search in the MSO97RT.DLL file for the id. These ids are defined in MSOBTNID.H in %RADBIN%\common\inc\office\v8. If you know of an icon id that is in this file that you can use, then use it (but please ensure nobody else is using it as well). If you don't have an icon, specify GuidOfficeIcon:msotcidNoIcon.

   There is also support for providing your own icons. To do this, you need to specify another guid, say GuidVJBmp instead of the Office one. You need a new guid for each bitmap resource you have in your package's dll. The bitmap resource should be **16*NumIcons** pixels wide and 16 pixels high, and the **background color** should be set to **RGB(0, 254, 0)**. The IconId that you choose should be the index of the icon in this bitmap, beginning at 1. (ie the first icon in this bitmap would be GuidVJBmp:1). If you specify your own bitmaps, then you need to add a BITMAPS_BEGIN – BITMAPS_END section which is described later on in this section.

5. **BUTTON Type**
   Ex. BUTTON
   BUTTON
   BUTTON
   BUTTON
   BUTTON
   This section determines the type of the button, either BUTTON (most common) or SPLITDROPDOWN.

6. **FLAGS**
   Ex: ,
   ,
   ,
   ,
   ,
   In this example, no flags are necessary so we just don't add any. For a description of the flags that can be used, look at the Command Bar UI spec on http:\\rad\specs. Common flags are TEXTCHANGES and DYNAMICVISIBILITY. These refer to the text being able to change during run mode (instead of remaining static), and the command being able to be hidden or shown respectively. The flags can be OR'd together to have more than one apply.

7. **BUTTON Text**
   Ex: "Cut"
   "Copy"
   "Paste"
   "Override Method"
   "View Properties"
   This is the text that shows up on buttons that are NOT in a menu off of the Main Menu bar.

8. **MENU Text**
   Ex: ,
   ,
   ,
   ,
   "Properties"
   This is the text that shows up on the item if it is in a menu off of the Main Menu bar. The reasoning behind this is that you can have 1 command with different text depending on where it is located in the menu structure (ie on a context menu versus a main menu). In this example, it makes sense to have the text "View Properties" on the context menu. However, if that item also showed up in a group in the View menu, we wouldn't want the text to be the same, since the user would see View/View Properties. Instead, by adding the menu specific text, the user will see View/Properties, which makes far more sense.

9. **TOOLTIP Text**
   Ex: ,
   ,
   ,
   ,
   ,
   Again, as in the MENU text situation, if you want the tooltip text to be presented differently to the user then you should add different text here. In this example, the tooltip text will be the same as the button text, so none is added.

10. **COMMAND Name**
    Ex: ;
    ;
    ;
    ;
    ;
    This is the command name of the item that will show up in the Customize and Keyboard Customization dialogs. If you want the text of this item to be presented differently in these dialogs, then you should add the text here. Otherwise, the BUTTON text is used.

2.2.1.4 COMBOS_BEGIN – COMBOS_END

This is the section in which a package defines its combo boxes. There are three types of combo boxes that a package can add to the shell. They are:

MRU Combo
This type of combo is created and filled by the shell on behalf of the package. The user is able to type in the combo, and the shell will remember up to the last 16 entries per combo. When the user selects something in the combo, or types in something new, the shell notifies the appropriate package. See the section on Command Contracts in the Command Bar UI spec for information on how this occurs.

Dynamic Combo
This type of combo is created by the shell on behalf of the package. The package is responsible for filling in the contents of this combo. The user is allowed to type in this combo as well as select items within it.

DropDownCombo
This type of combo is also created by the shell on the package's behalf. The package is responsible for filling in the contents for this type as well. However, the user cannot type anything into the edit field of this dropdown – it is simply a place where they can choose from a list of items.

The combo section takes 11 parameters:

1. **GUID:CmdId** - The command's id
2. **GUID:GrpId** - The primary group the combo resides in
3. **Priority** - The priority within that group
4. **CmdId** - This id is a secondary id that the shell will use to get the items that should appear in the combo. See the Command Bar UI spec for more information on how this occurs.
5. **Width** - This is the width the combo should be in pixels
6. **ComboType** - This is the type of the combo (as above)
7. **Flags** - This specifies the flags for the combo. These types are from the same location as those for buttons.
8. **Button Text** - This is the text the combo should take. It is really only necessary to add text to the tooltip, but there may be occasions where it is required to have text in all locations.
9. **Menu Text** - same as before
10. **Tooltip Text** - same as before
11. **Command Name** - same as before

2.2.1.5 BITMAPS_BEGIN – BITMAPS_END

This is the section in which a package defines its bitmap resources. This section is comprised of a minimum of 2 parameters, with no fixed upper limit:

1. **GUID:ResourceID**
   For each set of bitmaps, you need to specify a new guid. The resource id is the one that you've given it in your DLL.

2. **IndexNumber**
   Here you want to put the index number of all the icons you use from this resource. So if for example you had a bitmap resource number 400 in your DLL, and you used the icons at indices 1, 10, 17, and 20, then this entire entry would look like:
   GuidVJBmp:400, 1, 10, 17, 20;

2.2.2 CMDPLACEMENT_SECTION – CMDPLACEMENT_SECTIONEND

This section is used for placing commands, groups, or menus that are already defined as being in other primary locations. This is useful so as not to duplicate all of the required information that was added in the CMDS_SECTION. Our example doesn't need to use this section because all of the commands were already placed into their primary groups. However, if we wanted to reuse the commands we added and put them into other locations, we would do so here. There are only 3 parameters needed for each entry in this section. They are as follows:

1. **GUID:CmdGrpOrMnuId**
   This is the id of either the command, group or menu that you would like to place somewhere else. You must be careful here to follow the rules of how items are placed:
   - Commands are placed into groups ONLY. They cannot be placed directly onto a menu
   - Groups are placed onto menus ONLY. You cannot have a group within a group.
   - Menus are placed into groups ONLY. You cannot have a menu within a menu

2. **GUID:GrpOrMnuId**
   This is the id of the menu or group in which you want to place the item in parameter 1. Again, ensure you follow the rules of placement specified in the previous point.

3. **Priority**
   This is the priority of the item from parameter 1 with respect to items in parameter 2. The priority follows the same rules as before.

You are allowed to place commands into as many locations as you'd like, the only issue being that you need to have multiple command placement entries for the same command. If you do not need to use the CMDPLACEMENT_SECTION, then you don't have to include it in your text file.

2.2.3 VISIBILITY_SECTION – VISIBILITY_SECTIONEND

This section determines the static visibility of commands. You would put commands in here that you don't want to appear unless a specific package is active (either as a project or a document). This is strictly for Commands only – no groups or menu identifiers are allowed. Those commands that are not placed in here are set to visible by default. If you want the command to be visible for multiple packages you need to add it multiple times to this section. When the command tables are merged together, the multiple GUIDs will be OR'd to determine the final static visibility of the item.

This section takes 2 parameters:

1. **GUID:CmdId**
   This is the id of the command you want to control static visibility for

2. **GUID**
   This is the GUID of the package that you want the command to be visible for. When this GUID is active then the command will be visible.

2.2.4 KEYBINDINGS_SECTION – KEYBINDINGS_SECTIONEND

This section determines the keyboard mapping of the commands. Commands can have up to a 2 key binding associated with them (ie Ctrl-X + Ctrl-N), as well as any number of key bindings that will fire the command. This is useful for having multiple key bindings for 1 command, say Cut which could have both the Shift-Delete and Ctrl-X key bindings.

The section is comprised of 4 parameters:

1. **GUID:CmdId**
   This is the id of the command you want to assign a key binding to.

2. **GUID**
   This is the GUID of the Editor in which the key binding should be available.

3. **GUID**
   This is the GUID of the Emulation in which the key binding is available.

4. **KEYSTATE**
   This is made up in the following format:
   Key:Accel : Key:Accel
   The second Key:Accel is used if you want a two-key accelerator.
   The key part of it is the actual key in single quotes, such as 'A', or it could be a virtual key code, such as VK_F1. The Accel part of if is a combination of the following letters:
   A = Alt, S = Shift, C = Control, W = Windows Key. For example, if you wanted to add an accelerator for cut, such as CTRL+ALT+X, you would define it as 'X':CA

Parameters 2 and 3 are currently not implemented by the shell. For the time being, you can simply put guidVSStd97 in these locations. If you are adding a key binding that is a duplicate of an existing command, then you should use GuidKeyDupe for parameters 2 and 3.

2.3 Using the Binary Command Table

Once the textual representation of the command table is created, you need to then input the file as a parameter to the CTC program, which produces a CTO file. This CTO file is then included as a resource in your package. Note that the resource type **must** be defined as CTMENU in your RC file.

Note: In VJ98 and VB99, the CTO file is being stored in the shell's own IDE dll for now (in vj6ide.dll and vb7ide.dll respectfully). This will have to change in the future so that they become part of their own package's resources. These CTO files are being generated as part of the build process for each VB and VJ.

To make the new shell use your new menus and commands, you need to follow a few simple steps. As mentioned in the Command Bar UI document, your package must register its menus with the shell. This should be done in your package's DLLRegisterServer call. The package should place an entry in the registry in the HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\6.0\Menus section. The entry should look something like this:

Key Name	Key Data
CLSIDofPackage	c:\VisualStudio99\bin\VCPackage.Dll, 1000, 10

Note: In VJ98 and VB99, the shell is currently setting these up for VJ and VB. This will have to change eventually so that the appropriate package does the work instead.

When your package is unregistered, it should remove this key.

The data field is split up into three sections, separated by commas. The first section is the full path name to the file that contains the menu resource that the shell should use. The second section is the resource number of the menus in that file. **Note that this MUST be a number, not a name or identifier**. The final field is the version of the menu resource.

After registering your package, you should run the shell with the /merge switch to make it initialize its copy of the menus. Currently, you can do this by typing *VJ6 /merge*, or *VB7 /merge*.

3. Routing Commands to your Package

This is where you write real code. Commands are sent to your package through the IOleCommandTarget interface. This interface is documented in a number of places so I won't rehash it here as well. I will however point out some of the common pitfalls that people have fallen into when implementing it. I will also explain how the shell gets a hold of your command target so that it can pass you these messages.

3.1 IOleCommandTarget Interface

The definition for the interface is in DOCOBJ.H. There are only two methods in this class that you need to implement if you want to receive notification that your command was selected. These are QueryStatus and Exec.

3.1.1 QueryStatus

QueryStatus is where the shell will call you to determine the state of your command. This method was originally written to support getting status on a number of commands at once by sending in an array of command ids. However, because of the way the shell and Office Commandbars work, you can assume that the shell will only ever ask your package for status on 1 command at a time. NOTE – if your package is going to be used by anybody else other than the VS99 shell, this may be a bad assumption and you should implement QueryStatus as if it could get called with an array of commands. Be forewarned that it seems as if nobody (at least none of the packages we've seen) implements QueryStatus correctly when it comes to multiple commands. If you are going to do so and you delegate commands to another QueryStatus method, that you don't simply return from that call. You need to keep a list of all the commands that you sent as well as a list of commands that you responded to yourself. When done, you'll need to merge these lists and return them back to whoever called you.

One of the parameters to QueryStatus is a GUID. This is the GUID of the Command Group, such as GuidVJCmd in the previous example. You need to check to see if you know that GUID and can respond to the commands that are available in the command set defined by it. If you don't understand the GUID, you must return OLECMDERR_E_UNKNOWNGROUP. If you do understand the GUID, but don't understand (or haven't implemented) the command, you should return OLECMDERR_E_NOTSUPPORTED.

If you do understand both the GUID and the command, then you should fill in the prgCmds[0].cmdf field with the following information, as it applies to your command:

```
prgCmds[0].cmdf = MSOCMDF_SUPPORTED;
prgCmds[0].cmdf |= MSOCMDF_INVISIBLE;  // if the command should be invisible at the moment
prgCmds[0].cmdf |= MSOCMDF_LATCHED;    // if command is toggled on, pressed down look
prgCmds[0].cmdf |= MSOCMDF_ENABLED;    // if the command is enabled
```

If the command was set up with the TEXTCHANGES flag, then you should set the pCmdText->rgwz equal to the new text of the command, as well as setting pCmdText->cwActual to the size of the string.

3.1.2 Exec

Exec is somewhat simpler to implement than QueryStatus. When your package gets called on the Exec method, you need to again check the GUID and the command passed in to see if you understand them. If not, do the same as mentioned above for QueryStatus.

If you do understand the command, you simply do what you need to do and return when you are done. Your package is responsible for error detection and notification, so if an error occurs in the process of firing your command, you need to report it. The shell will assume that you have handled the command if you return anything but the error codes mentioned in QueryStatus.

For further information about the other parameters to Exec, such as the two VariantArgs and the nCmdexecopt, please consult the IOleCommandTarget spec or the Command Bar UI spec for special contracts that have been defined.

3.1.3 Sample QueryStatus and Exec implementations

The following is a sample implementation of QueryStatus and Exec that your package would be required to implement. This assumes that your package is only being used by the VS Shell, so that you only need to respond to QueryStatus one command at a time.

```
HRESULT CMyCommandTargetClass::QueryStatus
(
 const GUID     *pguidCmdGroup,
 ULONG          cCmds,
 OLECMD         prgCmds[  ],
 OLECMDTEXT     *pCmdText
)
{
HRESULT hr = NOERROR;
BOOL fHandled = TRUE;
BOOL fTextChanged = FALSE;
BOOL fEnabled = TRUE;
BOOL fDown = FALSE;
BOOL fInvisible = FALSE;
char szNewText[MAXPATHLEN];

ASSERT(cCmds==1, "Returning info for more than one item is NOT implemented yet.",;);

// check to see if these are commands owned by the shell or our own
if (IsEqualCLSID(*pguidCmdGroup, CLSID_StandardCommandSet97))
  {
  switch(prgCmds[0].cmdID)
    {
    case cmdidCut:
      fEnabled = FisCutEnabled();
      break;
    case cmdidUndo:
      fEnabled = Fundo();
      wsprintf(szNewText, "Undo %s", LpStrUndo());
      fTextChanged = TRUE;
      break;
    -
    default;
      fHandled = FALSE;
      break;
    }
  }


else if (IsEqualCLSID(*pguidCmdGroup, CLSID_VJPackage))
  {
  switch (prgCmds[0].cmdID)
    {
    case vjcmdOverrideMethod:
      fInvisible = FoverRide();
      fEnabled = FenableOverRide();
      break;
```

```
    default:
      fHandled = FALSE;
      break;
    }
  }
else
  {
  // delegate to inner object
  return InnerQueryStatus(pguidCmdGroup, cCmds, prgCmds, pCmdText);
  }

if (fHandled)
  {
  prgCmds[0].cmdf = MSOCMDF_SUPPORTED;

  // set the state flags
  if (fInvisible)
    prgCmds[0].cmdf |= MSOCMDF_INVISIBLE;
  if (fDown)
    prgCmds[0].cmdf |= MSOCMDF_LATCHED;
  if (fEnabled)
    prgCmds[0].cmdf |= MSOCMDF_ENABLED;

  // set the text for the item, if it has changed
  if (pCmdText && fTextChanged)
    {
    strcpyWfromA(pCmdText->rgwz, szNewText);
    pCmdText->cwActual = wcslen(pCmdText->rgwz);
    }

  hr = NOERROR;
  }
else
  hr = OLECMDERR_E_NOTSUPPORTED;

return hr;
)



HRESULT CMyCommandTargetClass::Exec
(
 const GUID     *pguidCmdGroup,
 DWORD          nCmdID,
 DWORD          nCmdexecopt,
 VARIANT        *pvaIn,
 VARIANT        *pvaOut
)
{
HRESULT hr = NOERROR;
BOOL fHandled = TRUE;

// check to see if these are commands owned by the shell or our own
if (IsEqualCLSID(*pguidCmdGroup, CLSID_StandardCommandSet97))
  {
  switch(nCmdID)
    {
    case cmdidCut:
      DoCut();
      break;
    case cmdidUndo:
      DoUndo();
      break;

    default;
      fHandled = FALSE;
      break;
    }
  }
else if (IsEqualCLSID(*pguidCmdGroup, CLSID_VJPackage))
  {
  switch (nCmdID)
    {
    case vjcmdOverrideMethod:
      FoverRideMethod();
      break;
```

```
    default:
      fHandled = FALSE;
      break;
    }
  }
else
  {
  // delegate to inner object
  return InnerExec(pguidCmdGroup, nCmdID, nCmdexecopt, pvaIn, pvaOut);
  }

if (!fHandled)
  {
  hr = OLECMDERR_E_NOTSUPPORTED;
  }

return hr;
}
```

3.2 How Commands are Routed to your Package

After writing your package's command target interfaces, you'll be able to receive commands. This happens in a number of ways. The basic routing mechanism the shell uses is defined in the CommandBar UI document. Your package will receive commands in one of the following ways:

- Tool Window. If your package is a tool window implemented in the proper manner, then it should have a command target that the shell will QI on. The shell will call your command target when your tool window is the active window.
  - If the tool window with focus is the Project Window, then it will route the command to the IVsUIHierarchy that is the common parent of the selected items. If the selection spans multiple projects, then the SID_SVsSolutionObject hierarchy receives the commands. The IVsUIHierarchy interface has QueryStatusCommand() and ExecCommand() methods that are analogous to the corresponding commands on IOleCommandTarget.

- Document Window. If your package creates a document window, the shell will also QI on the DocView object (either an instance of an IVsWindowPane object or an instance of an ActiveX DocObject) and get back a command target. If the DocView does not support the command, then the command will be routed to the IOleCommandTarget interface that can be retrieved via QI on the DocData object (this is an optional interface for DocData objects). If you want to bring up a context menu then you need to use the IOleComponentUIManager::ShowContextMenu() method. The shell will pass a pointer to the ComponentUIManager to DocView objects supporting the IOleInPlaceComponent interfaces. If you are not given a ComponentUIManager and you want to bring up a context menu, you need to get a pointer to the global ComponentUIManager by doing a QueryService on the shell's global service provider. This is how the context menus on hierarchies also work.

- Current Hierarchy (Project). The current hierarchy in the Shell is the project that owns the active document window or the hierarchy with selection in the ProjectWindow. The Shell will QI for IOleCommandTarget interface on the current (active) hierarchy. The hierarchy should support commands that are valid when ever the hierarchy is active, i.e. even if a project item document window has focus. Commands that should only apply when the focus is in the Project Window should be supported, instead, via IVsUIHierarchy. Examples of commands that should only be supported via IVsUIHiearchy are Cut, Copy, Paste, Delete, Remove, Rename, etc.

- When you bring up a context menu through the component UI manager, you pass in the command target which should be used to access your QueryStatus and Exec implementations.

- Your package itself should implement a command target that the shell will QI on from the IVSPackage interface.

4. Updating the Shell's UI

It is left up to the package itself to remember/know the state of the commands it can handle. The shell has no context on how to determine when a command becomes enabled or disabled, and as such the package must be responsible for informing the shell about this.

The shell receives all of its information about a command via the QueryStatus method that it calls on packages. There are a couple of cases in which the shell will call a package's QueryStatus method, one occurs when the user drops a menu or brings up a context menu. The other time this occurs is when the package requests that the shell should update the current UI. This is the interesting case for any commands that are currently visible to the user (such as cut/copy/paste on the standard toolbar).

The following is an excerpt of a piece of email that describes what needs to be done to inform the shell that the UI should be updated:

Q: When/Why do I want to update the menu/toolbar ui?

As a package with menu/toolbar items, you are responsible for updating the UI for those items whose state can change. This includes all the items that **you respond** to that are not created by you, such as Cut, Copy, Paste etc. A case in which you would need to tell the Shell to update the UI is if you respond to the Cut command and something you can cut was just selected/unselected (ie text in a buffer). The effect that you want to show is the Cut button on the Standard Toolbar become enabled/disabled. The Shell does not continuously poll every command to determine its state (huge performance hit), and therefore it is up to each package to tell the Shell that there is something that needs to be updated.

Q: My context menu and menu items seem to work fine. Do I still need to do this?

Yes. Whenever the user drops a menu or brings up a context menu, the Shell will do a QueryStatus on each of the commands on that menu to determine their state (toggled/enabled etc). That's why your items work in this case. However, any UI that is already up and visible to the user does not get updated by the Shell based on user actions alone. Also, any of your items that are only on menus or context menus could also be placed on a toolbar, either by your package or by user customization. Hence you must do this for any items that could change their state.

Q: So, how do I get the Shell to update the UI?

To get this to work, you need to QueryService on the Shell's Global Service Provider for SID_OleComponentUIManager with an IID of IID_IOleInPlaceComponentUIManager. You should get back an IOleInPlaceComponentUIManager pointer, which is AddRef'd on your behalf (you need to Release it when you're done with it). There's a method on this interface called UpdateUI that you need to call in the following manner: **pIOleInPlaceUIMgr->UpdateUI(0,FALSE,0)**. This will force the Shell to update the UI during its next idle loop.

As a side note, there is another way to get the IOleInPlaceComponentUIManager pointer. If you already have an IOleComponentUIManager pointer, you can just QueryInterface it with the aforementioned IID and you should get back an IOleInPlaceComponentUIManager pointer.

5. VB, VJ and Shell specific files

There are currently a number of files that are part of the VJ, VB and Shell builds which describe the menus. The following is a list of files and descriptions that are being used to do work in:

File	Description
Shell Specific	
VSSHLPRJ.H	Contains the shell's menu structure, ids, commands and groups. Use this file if you need to modify the shell itself. Most people won't need to do this.
VSSHLIDS.H	Contains definitions for the shell's groups and menus. Again, most people won't need to modify this.
VSSHLCMD.CPP	Contains the QueryStatus and Exec methods for shell specific commands.
STDIDCMD.H	Contains shared command ids. Don't add anything in here unless you are adding a command that can truly be used by other packages, or is a shell specific command.
VJ Specific	
VJMNUPRJ.H	Contains VJ's menu structure, commands and groups. Use this file if you need to add commands to VJ's package or menus.
VJMNUIDS.H	Contains definitions for VJ's groups, menus and commands. Use this file if you need a place to define any of the previous types.
VB Specific	
VBMNUPRJ.H	Contains VB's menu structure, commands and groups. Use this file if you need to add commands to VB's package or menus.
VBMNUIDS.H	Contains definitions for VB's groups and menus. Use this file if you need to create new ids for VB.
ICMDMENU.H	Contains definitions of VB's commands. Eventually this file should go away and become part of VBMNUIDS.H when the VB package works better.

6. Resources

There are a number of groups who have already implemented command targets and menu command tables, thus there are some examples already available. This document combined with the Command Bar UI spec on http:\\rad\specs, should be able to answer most, if not all of your questions. If there is something that hasn't been covered in here that you have questions about, feel free to send me, SteveSei, a piece of email regarding your questions.

EXHIBIT B

Exhibit immediately follows this page.

[Remainder of Page Intentionally Left Blank]

Command Bar UI

Authors: Derek Hoiem
Martyn Lovell
Bill McJohn
Steve Seixeiro

File: Command Bar UI.doc
Last Change: 08/04/97 8:18 AM
Printed: 03/04/99 11:04 AM

Section Contents

1. INTRODUCTION 1
1.1 SCENARIOS 1
1.2 REQUIREMENTS 1
1.3 JUSTIFICATION 2
1.4 ISSUES 2
2. FEATURE DESCRIPTION 2
2.1 USER EXPERIENCE 2
2.1.1 Hiding/Showing Commands 3
2.1.2 Default Toolbars 4
2.1.3 The Environment Default Toolbars 4
2.1.4 Product Specific Toolbars 4
2.1.5 Context Menus 5
2.1.6 Scrolling Menus 5
2.1.7 Docking Behavior 5
2.1.8 Toolbar Customize Dialog 5
2.1.9 Keyboard Customize Dialog 7
2.2 ARCHITECTURE 8
2.2.1 Contributing to the Command Space 8
2.2.2 Placing and Positioning Commands on Menus and Toolbars 10
2.2.3 Hiding and Showing Commands 10
2.2.4 Dynamically Changing Command UI 11
2.2.5 Handling Customization 11
2.2.6 Types of Commands 12
2.2.7 Command Routing 12
2.2.8 Command Contracts 13
2.2.9 Text Format of the Command Table 17
2.2.10 Resource Format 19

2.3 EXTERNAL DEPENDENCIES 24
2.4 INSTALLATION ISSUES 24
2.5 SCRIPTING/AUTOMATION CONSIDERATIONS 24
2.6 APPLICATION INTEROPERABILITY 24
2.7 INTERNATIONAL SPECIFICS 24
2.8 ACCESSIBILITY SPECIFICS 24
2.9 PERFORMANCE/CAPACITY REQUIREMENTS 24
2.10 TESTING PLAN 24
2.11 U.E. ISSUES 25
2.12 USABILITY TESTING PLAN AND ISSUES 25
2.13 FEATURES RESERVED FOR FUTURE VERSIONS 25

Abstract

This document describes the menu and command bar behavior, structure, and customization features of the Visual Studio environment.

Spec Owners/Contacts

Group	Owner
PM	DerekHo
Development	SteveSei
QA	EJugovic
UE	

Reviewers

Review group	Members
Working group	Derek Hoiem, Peter Loforte, Martyn Lovell, Bill McJohn, Philippe Nicolle, Steve Seixeiro
Primary reviewers	Chris Fraley, Michael Halcoussis, Doug Hodges
Secondary reviewers	

Review History

Date	Leader	Result
1/30/97	Derek Hoiem	For distribution to working group, and communication to DevStudio team on progress.
2/7/1997	Derek Hoiem	For distribution to primary reviewers.
2/13/1997	Derek Hoiem	For distribution to primary and secondary reviewers.

Revision History

Date	Author	Description
1/30/97	Derek Hoiem	Very rough draft. Basic scenarios and requirements for parity with DevStudio. Created headings (issues) for architectural requirements.
2/7/97	Derek Hoiem, Martyn Lovell, Steve Seixeiro	Major draft of Architecture section. Minor updates to User Experience section, although it is not complete.
2/7/97	Martyn Lovell	Added the results of today's meeting: Merged visibility tables, added bitmap tables, added some flags, added combo tables. Fixed context section.
2/13/97	Derek Hoiem, Martyn Lovell	Added new parts to the User Experience section, including environment menu items and preliminary discussion of default toolbars. Minor updates to the Architecture section for registry key info and the command table data to support keyboard emulation.
3/3/97	Derek Hoiem, Bill McJohn	Updates to the scenarios, issues, menus and toolbars sections. Added Bill McJohn's section on Command Contracts.
3/26/97	Derek Hoiem	Added link to command structure spreadsheet
4/29/97	Steve Seixeiro	Minor revisions to the Command Contracts section.
8/4/97	Derek Hoiem	Added Keyboard Customize dialog.

1. Introduction

The Visual Studio environment is a generic environment in which many tools can operate to provide the user a high degree of integration. To allow this high-degree of integration, all tools that are hosted, "packages", must participate in the Visual Studio environment's command environment. The environment defines standard commands and placement of those commands. Packages may add new commands and place them where they desire in the user interface, within certain guidelines. Commands are hidden and shown depending on the users context. The user may customize where commands appear and how they are accessed (via menu, keyboard, toolbar). Visual Studio's command bar appearance borrows from the Office suite.

1.1 Scenarios

A user opens a solution in Visual Studio that has many projects in it: a VB-created ActiveX control, a Java applet, an Active Server component created with Visual C++, and an InterDev website. Global commands from VB, VJ, VC, and InterDev are all shown in the command bars, but the InterDev website is the currently selected project, so in addition to the product-wide commands the InterDev project-oriented commands are shown. The user double-clicks an ASP file in the project and the file is openened in the text editor. Now, Visual Studio adds the text editor commands to the user interface. The user chooses "View/Preview in Browser" to see the website. Seeing that a control in a critical Java applet is positioned incorrectly, the user double-clicks a form of the Java project in the project window. Visual Studio hides the menus from the text editor and shows the commands from the form editor. Additionally, the InterDev project-oritented commands are hidden and the J++ ones are shown. The user then chooses the "Format/Align Right" command to correctly position the button in the Java applet. The user then remembers that the Java applet needs rebuilding so he chooses "Project/Build" (now enabled) and the Java applet is compiled into byte codes. Finally, the user chooses "Deploy" and seeing that the Java applet has a dependency on the InterDev project, the InterDev project appears in the browser with the newly updated Java applet.

A student in a computer lab is taking a class on Visual Basic. The computers in the university lab have the complete Visual Studio product suite installed since there are other students taking classes in other computer science subjects. The VB student double-clicks the Visual Studio icon and the IDE appears. The student chooses the File New command and is able to select a VB .EXE project. The new project appears in the project window, and only the VB commands are shown in the IDE. From this point on, the student sees only VB commands and he feels as if he is using only one product.

A user has a Perl script on a web server that she wants to edit. She loves the text editor in Visual Studio and decides to use Visual Studio for the task. She switches to Visual Studio and chooses the File/Open File command. Visual Studio shows the only the editor commands and the basic commands for window management and saving, closing, and opening files.

1.2 Requirements

- Enable many products (packages) to participate in a common development environment UI and contribute to the command structure.
- Provide a clear, predictable model for when commands appear/disappear and enable/disable. This includes hiding and showing of commands based on...

  Projects open in the current "solution"

  The project that has the selection in the project window

Any number of modes (such as debug, design, and run)

The active editor.

The file type in the active editor.

- Provide the ability for the environment to query the commands of a product without having to load its binaries.
- Provide users the ability to customize how commands appear in the user interface (menu, toolbar, keystroke, etc.).
- Provide products to put commands on toolbars that are not just buttons, but also combo boxes, buttons with drop-down lists, color picker, etc.
- Provide ability to localize command related info (menu text, tooltips, status text, command names, etc.) to any language without recompiling binaries.
- Leverage the existing Office command bar code.

1.3 Justification

This specification is a considered a core component of the Visual Studio environment. The primary justification for this feature is integration for all products into a single environment, and to provide a coherent user interface that scales to the user's current activity.

1.4 Issues

2. Feature Description

2.1 User Experience

Unlike Office, which has separate .exes for each product, the Visual Studio common development environment is a global environment into which individual products contribute. For the user, this means that Visual Studio has the potential to be very complex because it is the union of several products and their command sets. As a result, we must provide mechanisms in the environment to manage this complexity. The primary mechanism is the hiding and showing of commands based on the user's current activity. As a result, the user will see different commands appear as they perform various activities. This hiding/showing of commands is done to limit the commands in the user interface, otherwise a large number of disabled commands would appear, and menus would run off of the screen. Optimally, the user should not be aware that the command set available to them is changing. If users are confused by commands coming and going, then the design is not good enough.

To provide the ability for a high degree of customization, to save develoment time, and for a pleasant graphical look, the Visual Studio user interface will use the existing Office Command Bar mechanism for showing commands and tools in the user interface. Command Bars remove the distinction between toolbar buttons, menus, and their commands. All commands are mapped to buttons, which may have a text label or graphic, and in turn they may be structured into drop-down menus or rows of single buttons. Command Bars allow for a high degree of flexibility in the user interface, allowing for "traditional" menus and toolbar buttons to be mixed together. Despite this ability, the Visual Studio environment will show only two Command Bars by default: the main menu bar, and the default toolbar. Other Command Bars may appear and disappear based on the state of the application. Also, windows in the IDE may attach Command Bars

to their window edges. The main menu bar and default toolbar can be customized and even hidden, but the main menu bar cannot be deleted. The main menu bar and default toolbar are provided to establish a familiar and stable user interface, giving the user a sense of control.

2.1.1 Hiding/Showing Commands

Visual Studio environment has five factors that affect what commands are available in the user interface at any given time: *environment, product, project, editor,* and *file type*. The combination of these five factors is called the current *context*. When a command applies to the current context, the command is made visible in the user interface according to the rules described below; otherwise, the command is hidden:

> The *environment* is the overall development environment. Commands provided by the environment are always visible.
>
> A *product* may have commands that are always present after the product is installed. These commands do not need the context of a project in order to operate. For example, VIDs "View Links on WWW..." command does not need a project, just any arbitrary URL.
>
> A *project* is a discrete, "buildable", deliverable that is comprised of many files (source files). Projects are contained by solutions. Project commands will appear only for the currently selected project. Project commands for inactive projects are hidden from the user.
>
> An *editor* is a software component, with at least one window, that allows the user to operate upon one file in the project. Only one editor is active at one time. The active editor does not have to have the focus in order to be active. An editor can provide its own user interface but its command set must participate in the common environment. Commands that apply to the active editor are visible, but commands from any other editor are hidden.
>
> A *file type* is a variation of a file that can appear in an editor. An editor may have several file types that it operates upon, and it may customize its user interface depending upon the type of the file it is editing. It is up to the editor to decide whether commands that apply to the current file type in the active editor are shown.

Note that the hiding/showing of commands is separate from whether commands are enabled or disabled in the user interface. Enabling/disabling of commands is determined by the individual packages.

The two most important concepts to understand are:

1. Only one set of Project commands are shown in the UI at a given time based on the active project. The active project is defined as the project in the Project Explorer that is selected (or any file within it).
2. Only one set of Editor commands are shown in the UI at a given time.

So, at any given time, the UI will display the union of all environment commands, all product commands, the active project commands, and the active editor commands.

The environment defines a special command bar, the "main menu bar", which should be[1] the default location for commands that any package contributes to the user interface. The main menu bar is different from any other command bar because the environment uses it to control visibility of commands. All *other* command bars simply disable commands that are out of context, whether they are placed on a menu or a toolbar.

The environment defines a set of common commands structured onto the main menu. These commands are always visible, despite what packages are loaded into the environment. Packages may extend this set of commands. (See the *Architecture* section below.) However, the commands set from each product and the placement of their commands is the responsibility of each product team. The entire command set with default mnemonic and accelerator keys for Visual Studio is tracked in the spreadsheet below.

---

[1] I say "should be" because it may not be appropriate in all cases to place commands on the main menu bar. However, it should be a goal to keep almost all commands available in the UI on the main menu bar.

Visual Studio
Command Structure

The dark red commands in the spreadsheet are the commands provided by the environment[2]. The architecture allows packages to place commands anywhere in the user interface. Packages can add to the existing groups of environment commands, but package owners should confirm placement with the Visual Studio environment team before doing so. Placeholders are given for suggested locations where packages can define their own commands; however, these are given as guidelines. Owners of packages should communicate amongst each other to achieve consistency in commands placed elsewhere. For example, if a package would like to have a "Refresh" command, the owner should figure out the most appropriate place for it. If other packages would like to add a "Refresh" command, they should not place another Refresh command in the UI, but instead negotiate the placement of it with whoever else is using that command. Consistency of command placement is important is so that the menus remain stable across context switches.

2.1.2 Default Toolbars

In order to provide product identity and stability to certain functions in the UI, there are certain toolbars that will be present in Visual Studio by default. These default toolbars fall under three categories: Environment, Product, and Editor. All three types of toolbars are regular command bars, so they are present only until the user hides them or deletes them, and they are completely customizable.

2.1.3 The Environment Default Toolbars

The environment default toolbar provides commands that are shared among all products and that are core to the development environment. These commands include ones that apply to the solution and generic project operations (e.g. Save, Add Item, etc.) Products that plug into Visual Studio should not add or subtract to this toolbar, with one exception. If a product adds a new tool window, the window should be added to the list of tool windows (ones on the View menu). Otherwise, products should add their own toolbar. (See the next section.)

The standard toolbar takes up one full row at the top of the IDE screen. This is the only toolbar that the environment shows by default. Please see the command structure spreadsheet for the toolbar configurations.

Visual Studio
Command Structure

2.1.4 Product Specific Toolbars

Each product should provide a default toolbar that contains frequently-used and important commands for that product. Each product default toolbar should appear the first time Visual Studio is started after the product is installed.

Product toolbars are placed starting on the second row, below the standard toolbar. Subsequent toolbars are placed on the same row if there's room in the current screen resolution, otherwise they are moved to the next row.

---

[2] There are other commands provided by the environment, but are not seen in the UI. These are primarily used for convenience of other packages.

2.1.5 Context Menus

The environment defines a number of context menus. For more information, see the Visual Studio Command Structure spreadsheet:

Visual Studio
Command Structure

2.1.6 Scrolling Menus

This behavior is the same as Office. If the IDE window is maxmized, and a menu is long enough to "drop off" the bottom of the screen, the menu will display as many commands as possible along with a small scroll arrow. If the user has depressed the mouse button and floats over the scroll arrow, the menu will scroll downward so the remainder of commands may be shown.

2.1.7 Docking Behavior

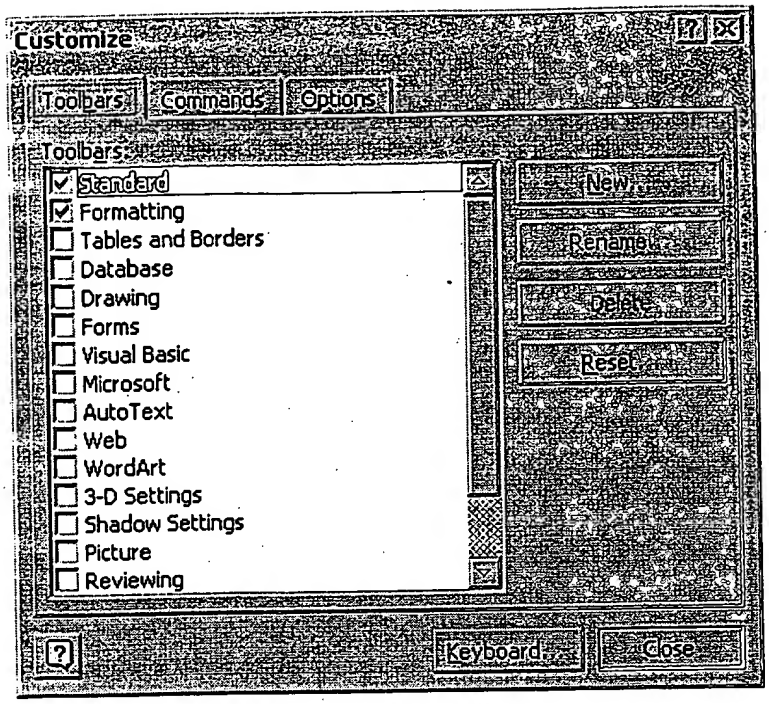
The docking behavior is the same as Office. Command bars are dockable to any side of the IDE main window.

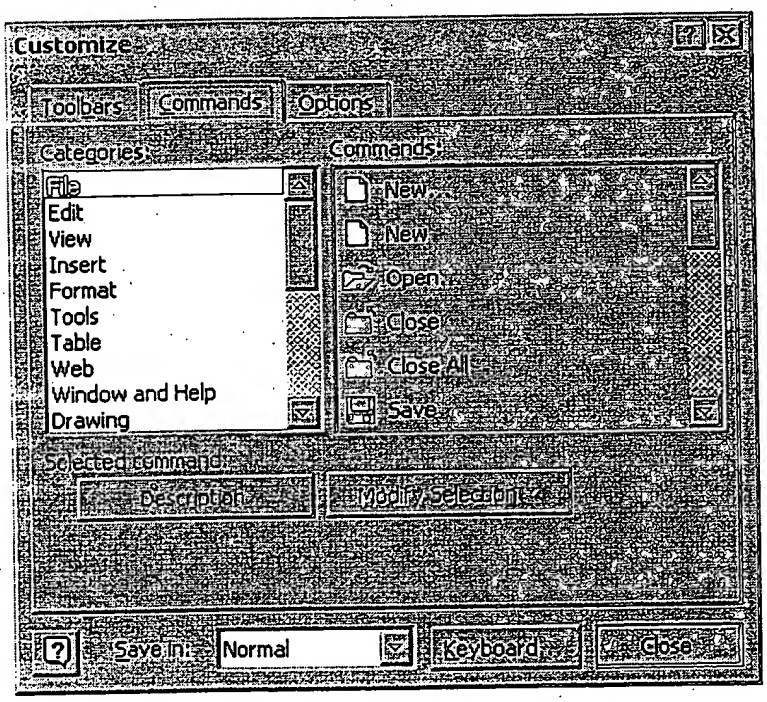
An added feature that we are investigating is toolbars docked to tool windows. However, create command bars that can be docked to the tool windows themselves. Command bars are not clipped by the IDE window and may exist "outside" of the IDE when they are undocked. Undocked command bars appear on a floating palette.

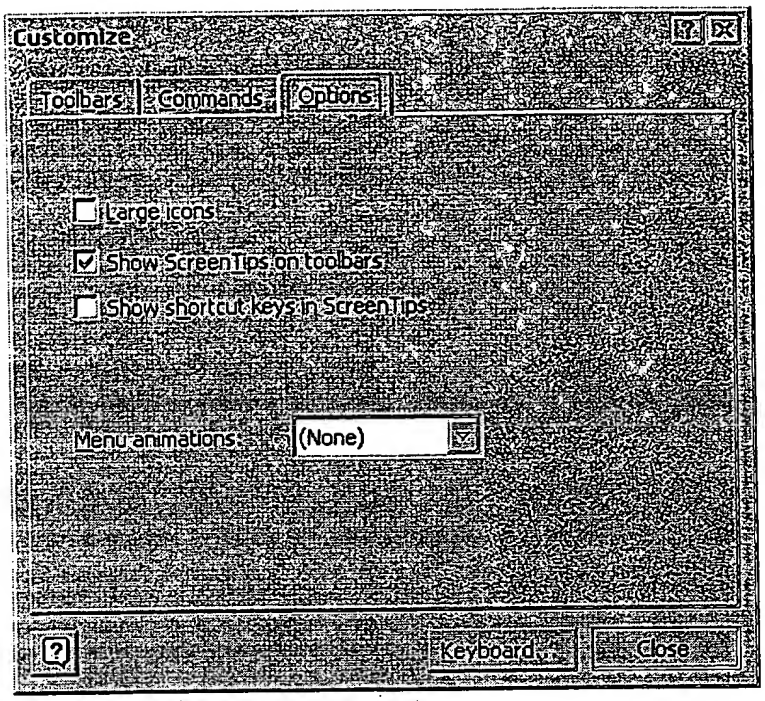
Any command that is not a button will turn into a button when it is docked sideways (on the left or right side of the IDE window).

2.1.8 Toolbar Customize Dialog

The Tools Customize Dialog is provided by Office and appears as follows. The Keyboard button launches the Visual Studio Keyboard customization dialog. See below.

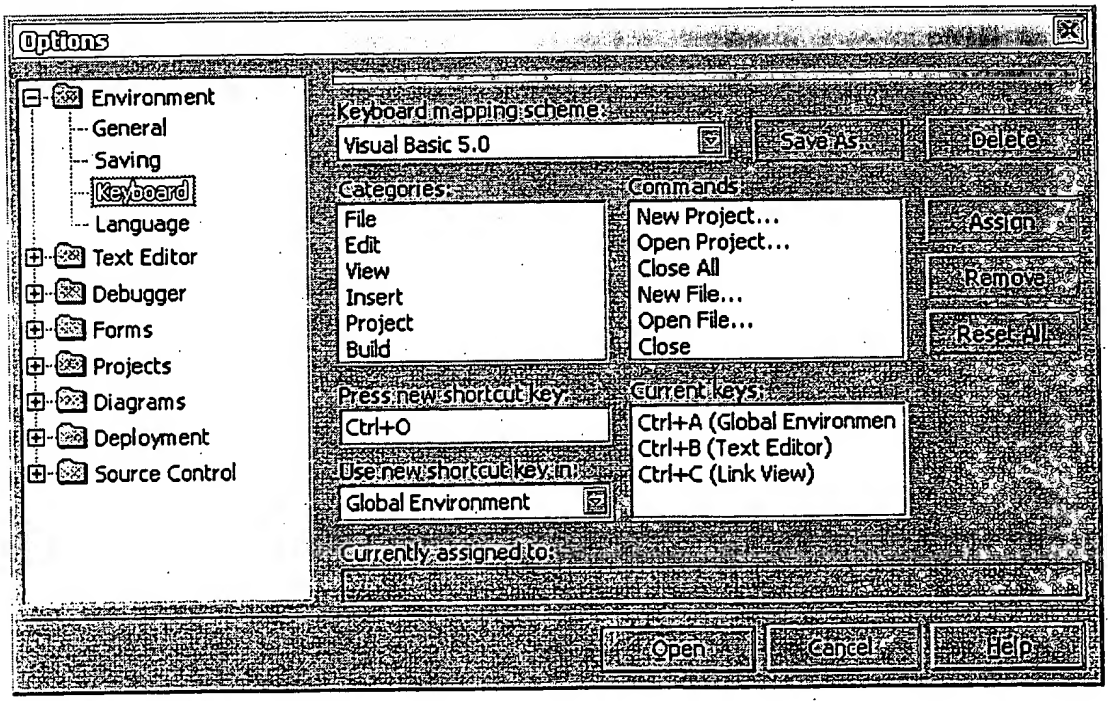

Customize
Toolbars
Commands
Options
Toolbars:
Standard
Formatting
Tables and Borders
Database
Drawing
Forms
Visual Basic
Microsoft
AutoText
Web
WordArt
3-D Settings
Shadow Settings
Picture
Reviewing
New...
Rename...
Delete
Reset
Keyboard...
Close

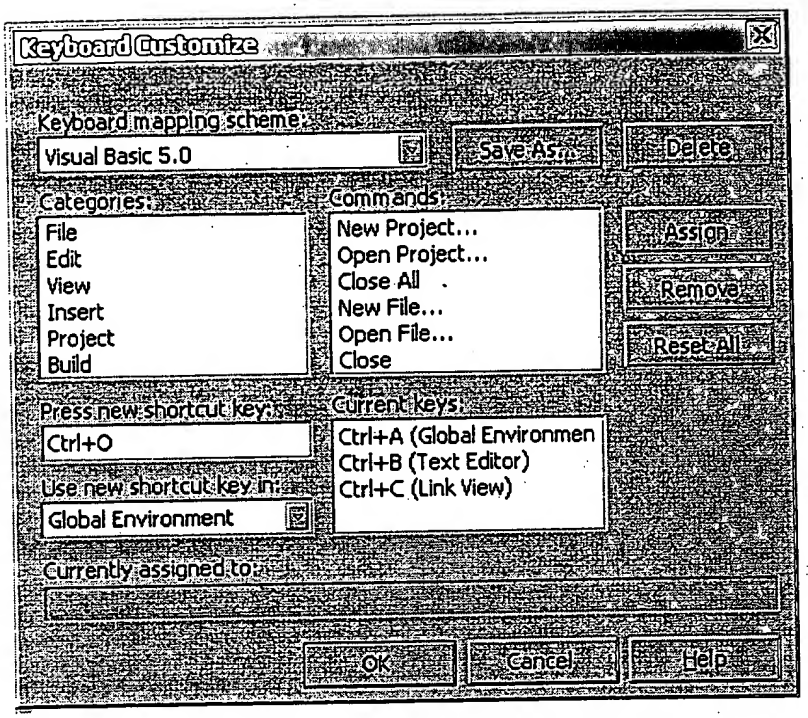

Customize
Toolbars
Commands
Options
Categories:
File
Edit
View
Insert
Format
Tools
Table
Web
Window and Help
Drawing
Commands:
New
New
Open...
Close
Close All
Save
Selected command:
Description
Modify Selection...
Save in:
Normal
Keyboard...
Close



2.1.9 Keyboard Customize Dialog

The Keyboard Customize dialog is the exact same one that appears in both the Tools Options dialog and off of the Keyboard button in the Toolbar Customize dialog. The Tools Options version is wrapped in the Tools Options frame.





The Keyboard Mapping Scheme loads an entire set of key bindings at once. By default, we will ship a Visual Studio, Developer Studio 97, and Visual Basic 5.0/6.0 keyboard schemes. The user can load a scheme by choosing it from the combo box. If the user makes any changes to the keyboard bindings using the Assign or Remove buttons, then the Keyboard Mapping Scheme combo shows "(untitled scheme)". The user can then give the untitled scheme a name using the Save As... button. The user is warned before they can save changes over an existing scheme name. Clicking the Delete button deletes the current scheme and loads the next scheme in the list.

The Categories list box lists all the top level menus, even if they are not current visible, as well as an "All Commands" category. The Commands list shows the commands within the currently selected Category. The Current keys listbox shows all keys currently bound to the selected command, including their scope.

The New Shortcut key edit field allows the user to press a key combination that will be used for the currently selected command. This field requires a modifier key (ctrl, alt, shift, Windows), but Alt cannot be used as a modifier by itself (since this activates the main menu bar). If the key the user pressed is already bound to a command, its name is shown in the Currently Assigned To field.

After pressing a shortcut key, the user can specify its scope, binding the key globally, or in a particular editor. This user can do this with the Use New Shortcut Key In dropdown. Once this combo is set, it is sticky for the duration of the dialog, otherwise Global is the default binding.

The Assign button adds the key and its scope to the Current keys list. The Delete button is active only if the selection is in the Current Keys list, and it deletes the selected key binding. Reset All returns all key bindings to the shipping defaults. (This is, in effect, the same as loading the default Visual Studio 98 keyboard scheme. However, the Visual Studio 98 scheme is not shown as loaded since the user may have changed it. As a result, the keyboard scheme list will show "untitled scheme" after pressing this button.)

2.2 Architecture

2.2.1 Contributing to the Command Space

This spec provides an overview of the architecture that Visual Studio uses for defining, placing, and showing commands. However, if you are developer that is implementing commands for your package,

then Steve Seixeiro's document on how to implement commands is very helpful. Steve's document is checked into the development tree, but

Implementing Commands

2.2.1.1 Packages

Packages' command data will be described in a text format, the "command table". Each command in the command table will be marked with meta-data including information identifying what package provided the command (a GUID), a command ID (a LONG), and other information. A package can use any GUID not just its own to define a command. (For more information, see the *Command Data Structure* section below.) The format is extensible, and allows for backwards compatibility. The command table is converted to a binary format which will include the data a resource[3]. The component will place this binary data in a custom resource of its .DLL. This scheme allows 1) the environment to add all menus/toolbars for the packages and let them be customized without the component being loaded 2) strings to be localized without recompilation of the package binary.

2.2.1.1.1 Installing

Upon install, the environment will create a "Master Command Database" from the binary data included in the individual packages. This database will be rebuilt every time a new package is detected.

Additionally, Visual Studio will keep track of installed packages and menu customization through the following registry key:

/Software/Microsoft/VisualStudio6.0

/Menus

Keyname	*Data*
{clsidPackage}	*"c:\VisualStudio98\bin\VCpackage.dll, ResourceNumber, Version"*

/MenusMerged

2.2.1.1.2 Uninstalling

The environment will maintain a list of known installed packages. When a package cannot be loaded, the environment will rebuild the Master Command Database and remove any customized commands that are no longer valid.

2.2.1.2 Add-Ins

Add-ins will use the environment's existing add-in interfaces and any interfaces that the various packages make available. (In other words, the object model of command bars, the Visual Studio environment, and any other products that plug into the environment.) They may also use the Package method, as described above.

---

[3] This process is similar to .ODL files which are converted to .TLB files with MkTypLib.Exe and then included in .DLLs (or .OLBs). Note, the binary files (and hence any .DLL with such a resource) may not always be compatible during the development process.

2.2.2 Placing and Positioning Commands on Menus and Toolbars

2.2.2.1 Groups

The environment uses combination of groups and priorities to position commands. Groups are logical groupings of commands that show up in the user interface. Within menus and toolbars, each group is denoted by a separator. The environment, by default, defines only a few groups and commands. (See the *User Experience* section above.) The command table format is flexible so that any package can add any group, command, or menu to any place in the user interface.

Every command is assigned to a primary group. This primary group controls the command's position in the main menu structure and in the Customize dialog. Each command can be in multiple groups (such as on main menu, on a context menu, and within a toolbar).

2.2.2.2 Priorities

To position commands within a group, each command has a priority. Similarly, to position groups within a menu or toolbar, each group has a priority. The priority is a 32-bit number, where zero represents the topmost position on the menu or in the group. By convention, the top sixteen bits will normally be used, leaving the bottom 16 bits to be used when commands must be "squeezed in" later.

2.2.2.3 Defining Top-Level Menus, Toolbars, and Context Menus

To define a top-level menu, you must first decide in which primary group the menu will appear and its priority. If necessary, you may wish to add a new group to an existing menu or toolbar, or you may create a new group in a new top-level menu, toolbar, context menu, etc. Once you have chosen or created a group, create an entry in the menus table in the command table (text file) which defines the menu itself. Finally, you should add any groups that the menu is to contain to the groups table and assign them appropriate priorities. You can then move the required commands into those groups. For more information, see the *Text Format* section below.

2.2.3 Hiding and Showing Commands

As described in the *User Experience* section, commands are hidden and shown based on the user's current context. Whether a command is hidden or shown depends on its flags noted in the Command Table.

Commands and menus are always visible if they are on toolbars, regardless of whether they are "in context" or not. However, when comamands are within the main menu bar, the system will show them only if they are relevant. Note that *menus* on the main menu bar, but not the command buttons on them, behave as if they were on a menu - that is, they are also context sensitive. Menus are never explicitly made visible or invisible. Instead, a menu is present only if a command on it are present. If no commands are present on a menu, then it is hidden. This rule is applied recursively.

2.2.3.1 Global level

To make a command appear always after a package is installed, set the "Dynamic Visibility" flag for the command.

2.2.3.2 Project level

To make a command appear only when a project of your type is active, set the "Project Dependent" flag for the command and list the projects to which the command applies.

2.2.3.3 Editor level

To make a command appear on when a certain editor is active, set the "Editor Dependent" flag and list the editors to which the command applies.

2.2.3.4 File-type level

To define a command that will appear only when a certain file type is used in an editor, follow the same procedure as defining editor commands. Keep in mind that each file type will have its own GUID, so it will work just like an editor-specific command.

2.2.4 Dynamically Changing Command UI

There are certain times when it is necessary to have commands that change their appearance or text at runtime. Here is how these cases are handled.

2.2.4.1 Enabling and Disabling Commands

To further delay the loading of binaries, packages can add a flag to determine whether a command is enabled or disabled by default. If a command is enabled, the user may choose it and the binaries for the package will load when the command is routed. All enabling and disabling from that point on is handled by the package itself.

2.2.4.2 Changing Command Text

If you wish to have different text for a command when it is on the main menu (versus another location), you can indicate this in the command table. However, if the command needs to change its text based on other context information, such as project name, then the command should set the "Text Changes" flag in the command table and respond to IOleCommandTarget->QueryStatus.

2.2.4.3 Changing Command Glyph at Runtime

This will not be supported. Sorry!

2.2.4.4 Determining the Context of UI Update

Sometimes you may want to know whether a command was activated from a menu vs. context menu, etc. The environment will handle context based differences between main menus and context menus. If the application wishes different updates to happen for different context menus, it should pass in a different IOleCommandTarget

2.2.5 Handling Customization

It is possible to customize the whole IDE without requiring any package to be loaded. The meta-data in the command table includes the information to allow this.

When users enter customization mode, they will be shown menus appropriate for the current context; however, the Command Well (list of all available commands in the IDE) will be filled with all the commands for all contexts. If Office allows, we will provide a customization option to see menus for all projects, file types, and editors (i.e. "Show All Commands"). This will allow users to access any command that is currently installed in the IDE despite its current availability.

Persistence of command customization is handled by the Office code. The customizations will be saved under HKEY_CURRENT_USER in the registry. Solution-specific customizations are not supported.

2.2.5.1 Customization Exceptions

Sometimes it is necessary to put restrictions on how commands can be customized. Here are the options that are allowed:

- Commands that cannot be customized: set the NOCUSTOMIZE flag in the command table.
- CommandBar-Only commands: set the NOKEYCUSTOMIZE flag. This means it cannot be bound to a key.

- Keyboard-only commands: set the NOBUTTONCUSTOMIZE flag. This means the command cannot be bound to a button or menu item.

2.2.5.2 Maintaining Customization During Package Installation

The environment knows when a new package has been installed (see the *Installation* section above). When this event is detected, the environment will scan the new packages for commands and groups that need to be added to menus and toolbars with saved customizations.

2.2.6 Types of Commands

The types of commands that a package may add are dictated by those basic types available in the environment. This means that there is a limited number of ways a command can present itself in the user interface. The list of supported command types is a combination of those from the Office DLL and new types the environment has added. These types include:

BUTTON	Basic buttons, with or without icons
MENUBUTTON	Used for File, Edit etc
DROPDOWNCOMBO	Such as the search/zoom combos
MENUCONTROLLER	Such as those found on the VB5 Standard Toolbar, AddNewProjectType
SPLITDROPDOWN	Such as the Undo/Redo buttons on the Standard Toolbar in Word
MRU Combo	Combo box that remembers what you typed into it
Dynamic Combo	Combo box that requires data to be loaded from package

2.2.6.1 Dynamic Command Lists (MRU, Window List)

We have the concept of dynamic groups. This type of group is used for MRU lists, the Window list, and OLE verbs on objects.

Text is only supported for commands that appear in dynamic lists.

Filling in the data for this group will require the package to load when the menu is dropped and we ask the package to load the commands.

Multiple packages can contribute to the same dynamic list.

2.2.7 Command Routing

We will employ a command routing architecture so that commands can be handled by different packages. There are five levels through which commands are routed.

0. Addins
1. Context menu
2. Focus Window (Tool Window)
3. Active MDI Child (Document)
4. Current Project
5. Startup Project
6. Global (load if package is not loaded)

Any command can be marked with flags indicating at what level the command will be handled. For commands that are unhandled, the environment will simply load the package.

When a command is routed from a context menu, it is first offered to the command target provided to the context menu, and then to the normal routing.

2.2.8 Command Contracts

The basic contract for handling commands through IOleCommandTarget is that the host will call QueryStatus to determine whether the command is supported, and (if it is supported), its state and text. The host calls Exec (with NULL input and output parameters) to perform the command.

This simple contract suffices to update and handle buttons; for more complex command UI (in particular, dropdowns), further communication is required. This document describes the contracts for various kinds of dropdowns.

QueryStatus is handled identically for all commands; the further communication is managed by calling Exec with certain parameters. The interpretation of these parameters depends on the specific command.

If the command target returns values in the output parameter, the caller is always responsible for freeing any resources allocated. Since this parameter is a variant, clearing the variant will suffice.

The remainder of this document describes how a command target should respond to calls to Exec with various patterns of parameters for certain predefined commands and for commands which the package defines in its meta-data as dropdowns.

Any combination of parameters not described below should be treated as invalid. (The command target should return E_INVALIDARG and not perform and action.) This will avoid undefined behavior if the contract for a command is extended.

2.2.8.1 Specialized Contracts

These commands are all associated with the command set GUID CMDSETID_StandardCommandSet97; they are defined in stdidcmd.h. With the exception of **cmdidMultilevelUndo** and **cmdidMultilevelRedo**, they are all supported by some existing component (either the DaVinci tools or Forms96).

2.2.8.1.1 cmdidZoomPercent

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

> The output parameter receives the current zoom percentage as an integer (VT_I4). Note that the command target should return an actual percentage, not one of the defined special values (see below).

Input: VT_I4

Output: non-NULL (optional)

Execution Option: OLECMDEXECOPT_DODEFAULT

> Set the Zoom value. The input parameter supplies the new percentage. The following special values, defined in stdidcmd.h, may be used: CMD_ZOOM_PAGEWIDTH, CMD_ZOOM_ONEPAGE, CMD_ZOOM_TWOPAGES, CMD_ZOOM_SELECTION, CMD_ZOOM_FIT (see code snippet of enum below). The output parameter (if non-NULL) receives the updated zoom percentage as an integer (VT_I4). Note that the command target should return an actual percentage, not one of the defined special values.

```
// Zoom -- zoom values are passed as integers (VT_I4) which
// represent the zoom percentage (e.g. 37 for 37%).  Special
// values are passed as negative numbers, from the enumeration
// below:
```

```
//
enum
    {
    CMD_ZOOM_PAGEWIDTH      = -1,
    CMD_ZOOM_ONEPAGE    = -2,
    CMD_ZOOM_TWOPAGES       = -3,
    CMD_ZOOM_SELECTION      = -4,
    CMD_ZOOM_FIT        = -5
    };
```

The command target will receive the special values only if it includes them in the list it returns for **cmdidGetZoom**. (In other words, a command target need only handle those special values it chooses to display.)

2.2.8.1.2 cmdidGetZoom

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives a list of zoom values to display in the dropdown; this list is expressed as an array of integers (VT_ARRAY | VT_I4) in the output parameter. These values may be percentages or the special values described under **cmdidZoomPercent**. The host should display the values in the order they appear in the array.

2.2.8.1.3 cmdidFonts

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives the font of the current selection as a BSTR (VT_BSTR).

Input: VT_BSTR
Output: non-NULL (optional)
Execution Option: OLECMDEXECOPT_DODEFAULT

Set the font of the current selection to the font specified in the input parameter. If the output parameter is non-NULL, it receives the updated font for the current selection as a BSTR (VT_BSTR).

2.2.8.1.4 cmdidFontSize

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives the font size of the current selection. This value is expressed as an integer (VT_I4) which represent the font size in twips. (One twip is 1/20 of a point.)

Input: VT_I4

Output: non-NULL (optional)

Execution Option: OLECMDEXECOPT_DODEFAULT

The input parameter supplies the new font size for the selection, expressed in twips. The output parameter (if not NULL) receives the updated value.

2.2.8.1.5 cmdidMultilevelUndo

Note that this command will typically be handled by the Undo Manager object.

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives the description of the action on top of the Undo stack, expressed as a BSTR (VT_BSTR). (This allows the host to display a menu item of the form 'Undo xxx'.)

Input: VT_I4

Output: NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

Undo the number of actions specified by the input parameter.

Input: NULL

Output: NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

Undo the top action on the Undo stack.

2.2.8.1.6 cmdidMultiLevelUndoList

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives a list of undo actions to display in the Undo dropdown; this list is expressed as an array of BSTRs (VT_ARRAY | VT_BSTR).

2.2.8.1.7 cmdidMultilevelRedo

The contract for cmdidMultilevelRedo is identical to the contract for **cmdidMultilevelUndo**, except that it applies to Redo rather than Undo. The command to get the list is cmdidMultiLevelRedoList, which works the same as cmdidMultiLevelUndoList.

2.2.8.1.8 cmdidObjectVerbList0

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives the IUnknown pointer of the currently selected object. The host uses this pointer to enumerate the verbs supported by the object and construct the Object Verb menu.

Input: VT_I4

Output: NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

Instruct the selected object to perform the verb specified by the input parameter.

2.2.8.2 General Contracts

These contracts apply to categories of commands. When a package registers a command, it can specify in the command meta-data that this command uses one of these contracts.

2.2.8.2.1 Component-filled DropDownCombo

A component-filled dropdowncombo gives the package a way to display strings and allow the user to select a string or type in a new string. A package needs to specify a secondary cmdid that the environment will use to get the list of strings to display in the combo box.

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives a string (VT_BSTR) to be displayed in the combo's edit box.

Input: VT_BSTR

Output: non-NULL (optional)

Execution Option: OLECMDEXECOPT_DODEFAULT

Perform the command using the value specified by the input parameter. The output parameter (if not NULL) receives the new value to be displayed in the combo's edit box.

2.2.8.2.1.1 Secondary Cmdid

Input: NULL

Output: non-NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

The output parameter receives a list of strings (VT_ARRAY | VT_BSTR) to be displayed in the dropdown list.

2.2.8.2.2 Host-filled DropDownCombo (MRU combo)

The typical example of a host-filled dropdowncombo is the DevStudio Find combo. The user types in strings, which the combo remembers and displays in the dropdown list.

Input: VT_BSTR

Output: NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

perform the command using the input parameter as the argument.

2.2.8.2.3 Split DropDown

A split dropdown is identical to a dropdown except that it also has a button portion, which performs the command. This dropdown may be of any of the three basic types of dropdowns (component-filled, host-filled, or hybrid); it uses the contract for its basic type, with the addition of:

Input: NULL

Output: NULL

Execution Option: OLECMDEXECOPT_DODEFAULT

Perform the action with no argument. (Corresponds to clicking the button side of the split.) Note that the environment will have no need to ask the component for the edit text for a split-dropdown.

2.2.9 Text Format of the Command Table

The Command Table is defined in a text file. The text file has three major sections: Commands, Command Placement, and Visibility. Each of the sections is denoted with Begin and End identifiers and each section may have subsections.

One header file should be included at the top of each Command Table text file:

#include "vsshlids.h" // defines the constants used for environment-defined groups and ids

For a complete example of a Command Table, see the VSSHLPRJ.H file in the source tree.

2.2.9.1 Commands Section

The Commands Section is defined between the CMDS_SECTION and CMD_END identifiers. This is the first section in the text file.

The Commands section has three subsections for defining menus, groups, and buttons. The text denoted with the comment delimiters are descriptive and should be replaced with the actual identifiers.

```
CMDS_SECTION PACKAGEGUID
   MENUS_BEGIN
      //GUID:MENUID, PARENTGUID:GROUPID, PRIORITY, BUTTONTEXT,
      COMMANDNAME;
   MENUS_END

   NEWGROUPS_BEGIN
      //GUID:GROUPID, PARENTGUID:MENUID, PRIORITY, FLAG;
   NEWGROUPS_END

   BUTTONS_BEGIN
      //GUID:CMDID,  PARENTGUID:PRIMARYGROUPID, PRIORITY, GUID:ICONID,
      BUTTONTYPEID, FLAG [ | FLAG ], BUTTONTEXT, MENUTEXT, TOOLTIPTEXT,
      COMMANDNAME;
   BUTTONS_END

   COMBOS_BEGIN
      //GUID:CMDID,  PARENTGUID:PRIMARYGROUPID, PRIORITY,
      SECONDARYCMDID, DEFAULTSIZE, BUTTONTYPEID, FLAG [ | FLAG ],
      BUTTONTEXT, MENUTEXT, TOOLTIPTEXT, COMMANDNAME;
   COMBOS_END

   BITMAPS_BEGIN
      //GUID:RESOURCEID, BITMAPINDEX, BITMAPINDEX, BITMAPINDEX. . . ;
   BITMAPS_END
```

```
CMDS_END
```

All numeric IDS are LONGs. Priority is a DWORD. (See the *Prority* section above for more info.) BUTTONTEXT, MENUTEXT, TOOLTIPTEXT, and COMMANDNAME are all string literals and an ampersand (&) may be used to denote a mnemonic key. The mnemonic key appears in the user interface as an underline only when the command is on a menu.

Valid flags in the Groups section are:

DYNAMIC	to denote a dynamic group
Others??	??

Valid flags in the Buttons section are:

NOCUSTOMIZE	This command cannot be customized
NOKEYCUSTOMIZE	This command cannot be assigned to the keyboard
NOBUTTONCUSTOMIZE	This command cannot be assigned to a button
TEXTCHANGESONBUTTON	??
TEXTCHANGES	Text of this command may be changed at runtime
DEFAULTDISABLED	Command shows up disabled until package is loaded
DEFAULTINVISIBLE	Command is hidden until shown by a project or editor
DYANMICVISIBILITY	Command will be asked whether it is visible whenever it is asked whether it is enabled.
REPEAT	Command can be repeated

2.2.9.2 Command Placement Section

The Command Placement Section is denoted between the CMDPLACEMENT_SECTION and CMDPLACEMENT_END identifiers. This section comes after the Commands section.

```
CMDPLACEMENT_SECTION
   //GUID:CMDID, GUID:GROUPID, PRIORITY
CMDPLACEMENT_END
```

2.2.9.3 Visibility Section

The Visibility Section is denoted between the VISIBILITY_SECTION and VISIBILITY_END identifiers. This section comes after the Command Placement section.

```
VISIBILITY_SECTION
   //GUID:CMDID, CONTEXTGUID
VISIBILITY_END
```

2.2.9.4 Key Bindings Section

```
KEYBINDINGS_SECTION
```

```
    // GUID:CMDID, EDITORGUID, EMULATIONGUID, KEYSTATE;
KEYBINDINGS_END
```

The KEYSTATE is a string literal containing a description of key used to invoke the command. The string literal should have the following format

VK:Modifiers:VK:Modifiers

VK is a VK code, such as VK_RETURN or a single quoted ASCII character, such as 'a'.

Modifier is a combination of unquotes characters indicating which modifier keys should be used for the command.

A = Alt

S = Shift

C = Control

W = Windows Key

For example, to describe a two key combination Ctrl-X, Ctrl-Shift-C the syntax would be as follows:

'x':C:'c':CS

2.2.9.5 Emulations Section

```
EMULATIONS_SECTION
    // EMULATIONGUID, EMULATIONNAME;
EMULATIONS_END
```

The EMULATIONNAME is a string literal containing the name of the emulation, e.g. "Emacs"

2.2.9.6 Editors Section

```
EDITORS_SECTION
    // EDITORGUID, EDITORNAME;
EDITORS_END
```

The EDITORNAME is a string literal containing the name of the emulation, e.g. "Text". This will be the name that appears in the scope list of the Customize Keyboard dialog.

2.2.10 Resource Format

As mentioned above, the Command Table will be converted to binary format which is included as a custom resource in each package DLL. The Master Command Database uses this same binary format.

Global Header

Offset	Size	Description
0	4	Unique Dword to identify data as a command table ("MLCT")
4	4	Version – version of the data format, not version of the software
8	16	GUID of package providing the table
24	4	Total size of all subtables and the table header, in bytes
28	4	Offset from the start of the global header to the first table

Then repeat this

Table Header (Offsets are from the start of the table)

Offset	Size	Description
0	4	Unique Dword to identify type of table
4	4	Size of this table, including its header (LTS)
8	4	Offset from the start of the table's header to the next table, or NULL if this is the last table
12	4	Offset from the start of the table's header to the table data

Table Data for Menus table (Unique Dword: "MENU")

This is a unique table (only one table per command table)

Offset	Size	Description
0	20	GUID:Dword Id of menu
20	20	GUID:Dword id of Group containing menu
40	4	Priority
56	N	Strings, separated by \0 String 0: Button text String 1: Menu text (empty implies same as String 0) [unused for menus] String 2: Tooltip text (empty implies same as String 0) [unused for menus] String 3: Command Name (for Macros/Automation)
		Align to Dword boundary
		Repeat

Table Data for Groups table (Unique Dword: "GRUP")

This is a unique table (only one table per command table)

Offset	Size	Description
0	20	GUID:Dword Id of group
20	20	GUID:Dword id of menu **or menu controller** containing group
40	4	Priority
44	4	Flags DYNAMIC
		Repeat

Table Data for Placement table (Unique Dword: "PLAC")

This is a unique table (only one table per command table)

Offset	Size	Description
0	20	GUID:Dword Id of group or command
20	20	GUID:Dword id of group

40	4	Priority
		Repeat

Table Data for visibility table (Unique Dword: "VISI")

This associates a command with a project, editor or file type. The command will be invisible when that context is inactive

This is a unique table (only one table per command table)

Offset	Size	Description
0	20	GUID:Dword Id of command
20	16	GUID of project, editor or file type lexer This can take special GUID values NOEDITOR: Makes the command visible when there is no editor NOFILETYPE: Makes the command visible when the file type has no lexer.
		Repeat

Table Data for Commands table (Unique Dword: "CMMD")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID:Dword Id of command
20	20	GUID:Dword id of primary group
40	4	Primary group priority
44	20	GUID:Dword - Icon group, icon id
64	4	Button type
68	4	Flags: TextChanges NoMenu NoKeyCustomize NoButtonCustomize Repeat TextChangesOnButton DynamicVisibility DefaultDisabled DefaultInvisible
72	N	Strings, separated by \0 String 0: Button text

		String 1: Menu text (empty implies same as String 0) String 2: Tooltip text (empty implies same as String 0) String 3: Command Name (for Macros/Automation)
		Align to Dword boundary
		Repeat

Table Data for Combos and dropdowns table (Unique Dword: "COMB")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID:Dword Id of command
20	20	GUID:Dword id of primary group
40	4	Primary group priority
44	4	Secondary Cmdid
48	4	Combo default width
52	4	Button type
56	4	Flags: NoMenu NoKeyCustomize NoButtonCustomize
60	N	Strings, separated by \0 String 0: Button text String 1: Menu text (empty implies same as String 0) String 2: Tooltip text (empty implies same as String 0) String 3: Command Name (for Macros/Automation)
		Align to Dword boundary
		Repeat

Table Data for Bitmaps (Unique Dword: "GLYP")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID:Dword Id of initial bitmap
20	20	Count of bitmaps
40	4	First bitmap/bitmap strip resource id
		Repeat

Table Data for Key Bindings (Unique Dword: "KEYS")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID:Dword Id of command
20	16	GUID of Editor
36	16	GUID of Emulation
36	1	First Key
37	1	First Modifiers Bit 0 = Shift Bit 1 = Control Bit 2 = Alt Bit 3 = Windows Bits 4 to 7 are all 0
38	1	Second Key or 0 if none
39	1	Second Modifiers (see above)
		Repeat

Table Data for Emulations (Unique Dword: "EMUL")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID of Emulation
20	N	Emulation Name
		Align to Dword boundary
		Repeat

Table Data for Editor (Unique Dword: "EDIT")

This is a unique table (only one table per command table)

The data below is repeated

Offset	Size	Description
0	20	GUID of Editor
20	N	Editor Name
		Align to Dword boundary
		Repeat

2.3 External Dependencies

Obviously, we have a dependency on the Office command bar DLL. Office is planning an "Office97a" release. We would like to make any new requirements known to them and, if they incorporate these requirements, we would use this code base.

We would like our command bar UI to be as consistent as possible with IE4. However, IE is not going to be using the command bar code from Office.

2.4 Installation Issues

After each install/uninstall of packages or the environment, the master command database will be rebuilt.

2.5 Scripting/Automation Considerations

The Command Tables are where commands are given their command name that is used in scripting.

2.6 Application Interoperability

2.7 International Specifics

As mentioned above, all command meta-data will be maintained in a localizable resource file as part of the component.

2.8 Accessibility Specifics

2.9 Performance/Capacity Requirements

None known at this time.

2.10 Testing Plan

The AlexChi The test plan for the command UI can be found on
http://vbqa/vegasst/sharedide/ide/commandbarui.doc

2.11 U.E. Issues

2.12 Usability Testing Plan and Issues

2.13 Features Reserved for Future Versions